

I hereby certify that this correspondence is being deposited with the U.S. Postal Service with sufficient postage as First Class Mail, in an envelope addressed to: MS Amendment, Commissioner for Patents, P.O. Box 1460, Alexandria, VA 22313-1460, on the date shown below.

Dated: 4/17/06 Signature: Jesse Ryan

Docket No.: BBNT-P01-320  
(PATENT)

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE**

Re Patent Application of: Donaghey

Application No.: 09/536191

Art Unit: 2668

Filed: March 27, 2000

Examiner: Tran, Phuc H

For: Systems and Methods for Communicating in a  
Personal Area Network

**DECLARATION UNDER 37 C.F.R. § 1.131**

I, Robert J. Donaghey, do hereby declare as follows:

1. I am the inventor of the subject matter claimed and described in the above referenced patent application.
2. I discussed the legal meaning of the terms "conceived" and "reduced to practice" with my attorney. Based on my understanding of those terms derived from that discussion, I conceived of and reduced to practice, in the United States, the subject matter of the pending claims prior to June 25, 1999.
3. The technical documents attached as Exhibit A, dated January 11, 1998, and Exhibit B, dated December 12, 1997, were created prior to June 25, 1999. These documents include architectural details and software specifications illustrating embodiments of the present invention as claimed in claims 1-31 and 36-43. The documents illustrate exemplary embodiments of the invention and are not intended to narrow the scope of the claims.

Portions of computer source code, attached as Exhibit C, was written and executed prior to June 25, 1999. Dates, comments, and copyright information have been redacted from the source code. The attached portions of the source code illustrates the claimed portions of devices and means in a network as described in claims 1-20 that carry out the methods described in claims 21-31 and 36-40. The source code further implements a protocol as

described in claims 41-43. The code illustrates one embodiment of the invention and is not intended to narrow the scope of the claims.

4. In particular, Exhibit A describes:

- a. A network comprising a hub device configured to generate a token and broadcast the token on the network, at least at sections 1-2 of Exhibit A.
- b. At least one peripheral device configured to receive the token broadcast by the hub device, at least section 2 of Exhibit A.
- c. At least one peripheral device configured to determine whether the token identifies the peripheral device, at least at section 2 of Exhibit A.
- d. At least one peripheral device configured to analyze the token to determine a size and direction of a current data transfer when the token identifies the peripheral device, at least at section 2 of Exhibit A.
- e. At least one peripheral device configured to transfer data to or receive data from the hub device according to the determined size and direction of the current data transfer, at least at sections 2-3 of Exhibit A.
- f. A communications protocol used in a network connecting a hub device to at least one peripheral device, the communications protocol having a plurality of frames comprising a beacon that marks a start of one of the frames, at least at section 3 of Exhibit A.
- g. A network operating according to a communications protocol having a plurality of alternating token slots and data transfer slots, at least at section 3 of Exhibit A.

as well as related networks, devices, device means, memories, methods, and protocols associated therewith.

5. Exhibit B includes additional architectural details and software specifications related to the invention as claimed in the present application.
6. The attached source code in Exhibit C includes a procedure *Simulate\_Hub()* which includes code for generating and broadcasting a token (see, for example, code associated with the switch cases *TOKEN* and *XMIT*). The source code also includes procedures for receiving the token broadcast by the hub device, determining whether the token identifies the peripheral device, analyzing the token to determine a size and direction of a current data transfer when the token identifies the peripheral device, and transferring data to or receiving data from the hub device according to the determined size and direction of the current data transfer (see, for example, code associated with the switch cases *TOKEN* and *XMIT* within the *Simulate\_Pea()* procedure). Applicant notes that the code allows for determining a size and direction of a current data transfer by determining a stream identified in a token and determining a size and direction of a current data transfer associated with the stream (see, for example, code associated with the *if(reading\_token)* code block in the *Simulate\_Pea()* procedure, and stream assignments under the heading “--- Stream Assignments ---”).

The procedures *Simulate\_Pea()* and *Simulate\_Hub()* further allow for broadcasting the token during a token slot and communicating data between an identified peripheral and the hub on an identified stream during a data transfer slot, and for providing a beacon to mark a start of one of the frames (see, for example, the respective *switch* statements and *BEACON* switch cases of these procedures).

7. **Reduction to practice**

The code portions of Exhibit C provide evidence of the reduction to practice of systems and methods of the claimed invention. In particular, the code was executed before June 25, 1999 on processors of a hub device and a peripheral device, both having memories, in a network to transfer data between the hub device and the peripheral device.

8. I assert that all statements made of my own knowledge are true, and that all statements made on information and belief are believed to be true. I also understand that willful

false statements and the like are punishable by fine or imprisonment, or both (18 U.S.C. § 1001) and may jeopardize the validity of the application or any patent issuing thereon.

Dated: April 6, 2006

Robert J. Donaghey  
Robert J. Donaghey

## Exhibit A



---

# BodyLAN Link Layer Driver

## Architectural Details

---

Document number:      BodyLAN-32-01  
Last modified:        January 11, 1998  
Author:                Bob Donaghey and Rod Owen

### Table of Contents

<b>1. Introduction</b>	<b>2</b>
<b>2. Data Structures</b>	<b>2</b>
2.1 Streams Model	2
2.1.1 MAC Numbers	3
2.1.2 Stream Numbers and Multiplexing	3
2.2 Data Structures	4
2.2.1 Tokens and Data Buffers	4
2.2.2 Current Data Descriptor Rings	5
<b>3. The Hardware Dynamics</b>	<b>6</b>
3.1 Introducing the TDMA Plan	6
3.2 How PEA and Hub Interact with the TDMA Plan	7
3.2.1 How Hub Structures Relate to the TDMA Plan	8
3.2.2 How PEA Structures Relate to the TDMA Plan	9
<b>4. Software View of Hardware Setup (Static View)</b>	<b>10</b>
4.1 Stream Basics	11
4.2 How the Hub Sets up Descriptor Rings for Streams	11
4.3 How PEAs Set up Descriptor Rings for Streams	12
<b>5. Descriptor Rings for Hubs and PEAs (Dynamic View)</b>	<b>12</b>
5.1 The Hub's Dynamic View of the Descriptor Rings	12
5.2 The PEA's Dynamic View of the Descriptor Rings	13
5.3 Scheduling	14

## 1. Introduction

This document describes the static and dynamic features of the principal software data structures used in the dual-ported memory interface shared by the hardware and the link layer driver for BodyLAN. It thus inherently updates the "March scheme" described in "Data Structure Modifications for the BodyLAN Architecture," Document No. BodyLAN-18-00, by Rick LaRowe, and reflects, abstractly, "BodyLAN Hardware-Software Memory Interface," Document No. BodyLAN-13-04, by Bob Donaghey. Also, see the document BodyLAN-32-00, Sept. 25, 1997, for a detailed explanation of how the hardware instantiates the multiplexed communications streams.

The link layer driver (LLD) understands the BodyLAN hardware memory map and is responsible for initializing and updating the BodyLAN data transfer memory as it delivers and receives data blocks for the link layer transport, which sits above the driver layer in the software architecture. The LLD also handles interrupts by the hardware.

Hubs regularly transmit a TDMA beacon that PEAs can use to synchronize their TDMA clocks. After the beacon, the Hubs broadcast tokens that are received by all PEAs. Then one PEA and the Hub either transmit or receive data. The types of communications between them include beacons, token commands, tokens, data, and communications about data.

## 2. Data Structures

A streams abstraction is implemented around a core set of data structures that interface between the BodyLAN digital control logic and the host CPU processor. These data structures also play a critical role in the behavior of the finite state machines controlled by the TDMA plan.

### 2.1 Streams Model

The PEAs are arranged like spokes around the Hub, in a star topology. An abstract streams model was chosen to multiplex the communications between the Hub and PEAs. The general idea is that there is a relatively small set of predefined streams known to all BodyLAN devices, and all communications occur over those streams.

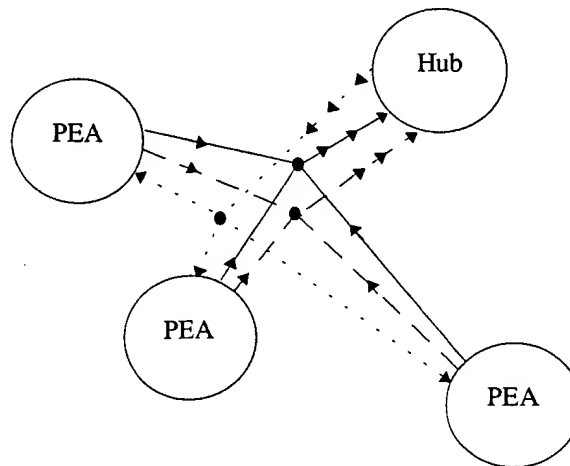


Figure 1: Multiplexed Streams

Logically, a stream is a one-way communications link between one PEA and its Hub. For example, there might be a network of three PEAs and a Hub, with three Hub-to-PEA streams, and six PEA-to-Hub streams, as shown in Figure 1; at the Hub, stream links from all of the PEAs are logically combined or multiplexed onto a single stream link, resulting (in this example) in two multiplexed streams in to, and one out of, the Hub.

### 2.1.1 MAC Numbers

MAC (media access) numbers are used to identify the Hub and PEAs, and they could also be used to identify virtual PEAs within any one physical PEA. Stream numbers are used to identify communication channels for specific functions, e.g., for data and control. The MAC and stream number are combined into a token, which is broadcast by the Hub and received by PEAs. The token's two parts are used to identify the PEA and the stream, and hence, indirectly, the correct data buffer.

### 2.1.2 Stream Numbers and Multiplexing

The MAC and stream numbers are instrumental in multiplexing communications. If it were not for the MAC number, which determines which PEA is meant to receive or send the communication, the streams could not be multiplexed. Each PEA uses one array to record which MAC it listens for, and multiple arrays to record which streams it uses..

Figure 2 shows a MAC array of 128 entries, and a stream array of 16, with streams 0 to 2 and 3 to 7 available for use for PEA-to-Hub communications for small and large data buffers, respectively; and streams 8 to a and b to f available for use for Hub-to-PEA communications for small and large data buffers, respectively.

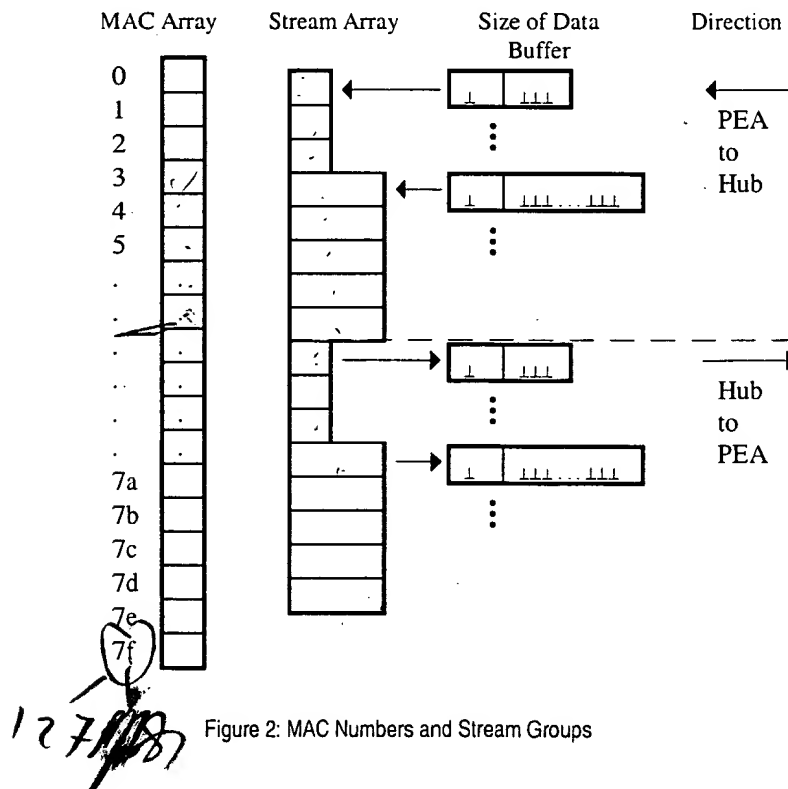


Figure 2: MAC Numbers and Stream Groups

The stream arrays, like the MAC array, are structures in the PEA's memory, but they the Hub controls which MAC and stream number it uses for a particular communication, and when it uses them. Thus, in Figure 3, which suggests how streams are multiplexed, the representations of these arrays next to the Hub is not meant to suggest that the arrays are physically present in the Hub.

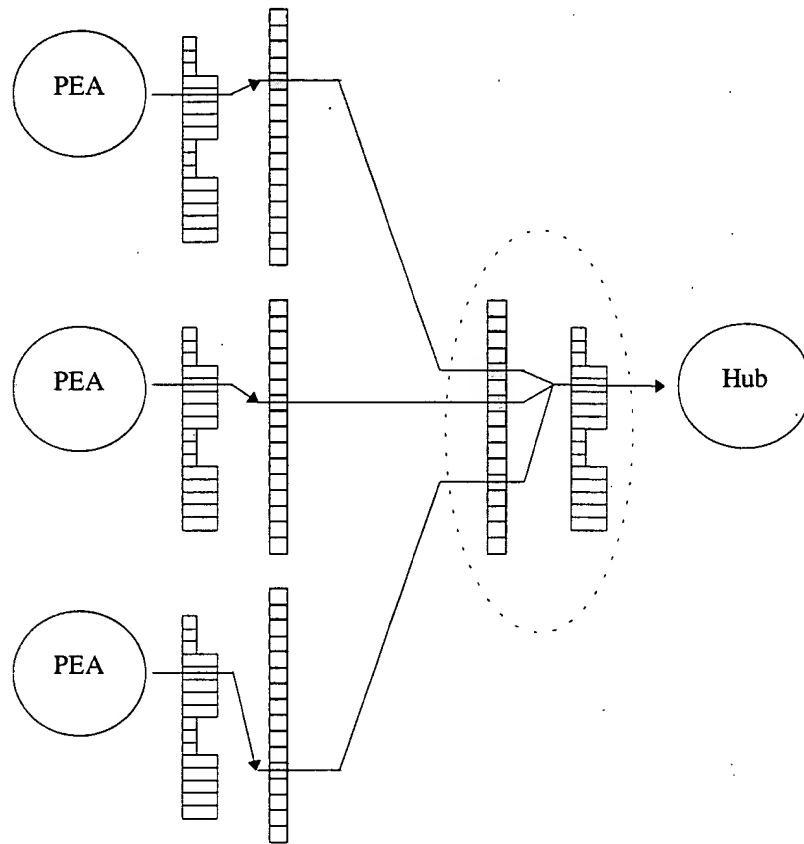


Figure 3: Multiplexing via MAC and Stream Arrays

## 2.2 Data Structures

### 2.2.1 Tokens and Data Buffers

Figure 4 shows an abstract representation of a token and data buffer, as well as a schematic representation. Notice that the schematic representation shows the data block, which includes both the token and the data buffer in a single combined structure, as these two objects always appear together in this design.

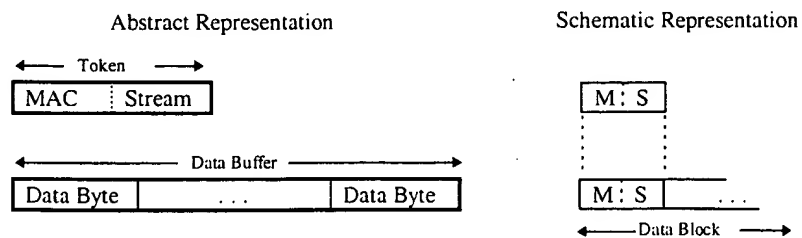


Figure 4: Token, Data Buffer, and Current Data Block

### 2.2.2 Current Data Descriptor Rings

The Hub and PEAs use linked lists whose elements point to the data blocks. These lists may be circular rings or linear chains whose final link points to itself. An alternative final element in descriptor rings has a 0 instead of a pointer to itself. This keeps the hardware from updating the current link pointer, achieving the same effect as a self-pointer while giving the software better control. These rings and chains are shown in this document as abstract drawings and schematics.

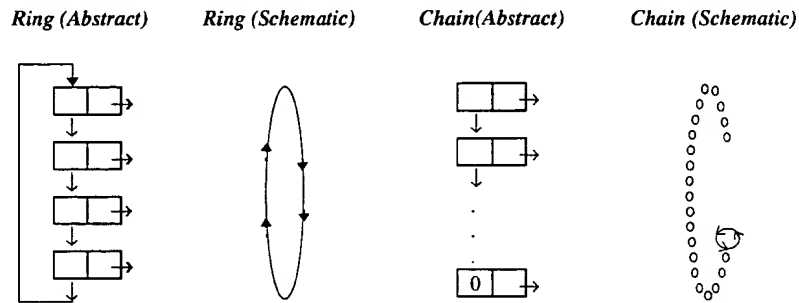


Figure 5: Two Types of Linked Lists

A pointer at the top of the rings and chains (see Figure 6) points into a specific link that marks the current element of the ring or chain. In the case of rings, the trail of pointers is followed in a circle, so that the finite state machine that traverses the linked list just keeps cycling around through all of the links. In the case of chains, the trail of pointers is followed until the last link, shown as "0" in the figures, is reached, at which point the end of the chain has been reached and the finite state machine thereafter cycles in a "ring of one," staying at the final link forever if its current-link pointer is not adjusted by software.

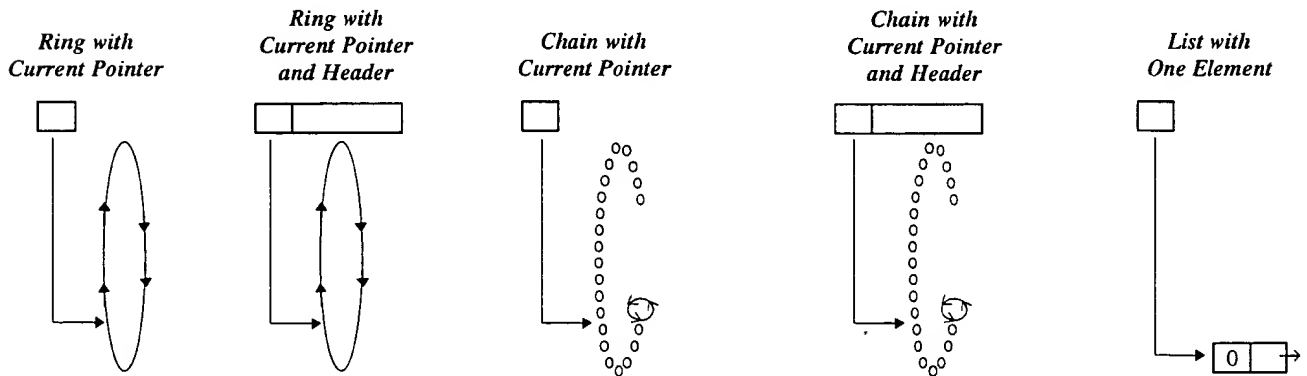


Figure 6: Types of Pointers into Linked Lists

An instance of a current pointer and header is seen in the *Descriptor Ring Header*, the highest-level data structure describing the attributes of a stream. Its fields include a current pointer known as a *ring locator*, which identifies a specific entry in the ring; *flags* (direction of the stream, CRC and interrupt enable) that control the behavior of the stream; a *length* field for the data contained in the ring's data buffers, and a receive/transmit *counter* for the stream.

### 3. The Hardware Dynamics

From the hardware's point of view, the relevant registers and memory locations in the Hub and PEA are modified in response to TDMA commands consisting of beacons, token commands, Xmits (with prior waits), and an "end" (return to beginning of plan). For a given BodyLAN network, these comprise one common, shared *TDMA program or plan* that executes over and over and is used by the hardware to write into and read from memory locations, while the software may also be reading and writing many of the same locations.

#### 3.1 Introducing the TDMA Plan

The Digital Control Logic manages a high-level *TDMA* (Time Division Multiple Access) structure as shown in Figure 7. Here, along the top line, each TDMA byte is represented by a B (for beacon), an X (for a one-byte transfer), or a T (for a token). Related functional groupings are in the boxes beneath. This structure is a *TDMA Token Plan*. Its purpose is to tell each PEA when in real time to expect the beacon and the dynamic bandwidth assignment tokens from the Hub. The TDMA plan also controls the maximum lengths of the data blocks.

There are four token commands ( $T_0$ ,  $T_1$ ,  $T_2$ , or  $T_3$ ), each of which may be followed by a different number of Xfer commands, but always the same number for the same command. The hardware uses the first six Xfers following a token command to transmit the token. Then, typically one PEA and the Hub either transmit or receive a byte of data as the hardware processes each remaining Xfer command following the token. Not all of the Xfers need to be used to transfer data: any left over are treated as no-ops to mark time. Similarly, all other PEAs not involved in the data transfer use all of the Xfers to mark time while waiting for the next token.

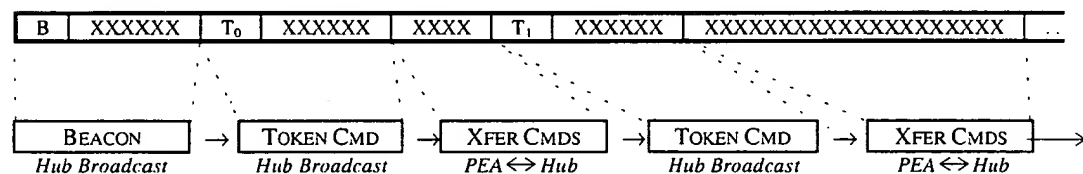


Figure 7: Byte-Oriented and Representational TDMA Token Plan

In summary, the basic types of "instructions" communicated over the BodyLAN network are:

- **Beacons** — Mark the start of each TDMA frame and are used by the Hub and PEAs to establish synchronization to the network.
- **Token Commands** — Tell all PEAs within listening range to expect to receive a token, the contents of which are the MAC and stream numbers for the data transfer to follow. The token commands also tell the Hub which token plan ring to use ( $T_0$ ,  $T_1$ ,  $T_2$ , or  $T_3$ ).
- **Xfer Commands** — Notify the PEA specified by the token to transmit or receive a byte of data. Six Xfers send the beacon or token; any remaining after the token are used to transfer data buffers (whose length must not exceed the number of Xfer commands).

For more information on TDMA structures, see "BodyLAN: Low-Power Wireless Communications Protocol Overview," Document No. BodyLAN-27-00.

### 3.2 How PEA and Hub Interact with the TDMA Plan

This section describes how the finite state machines of the PEA and Hub interact. The purpose is to show which stage the PEA is at vis-à-vis the Hub. The following two sections describe how these state machines interact with their data structures.

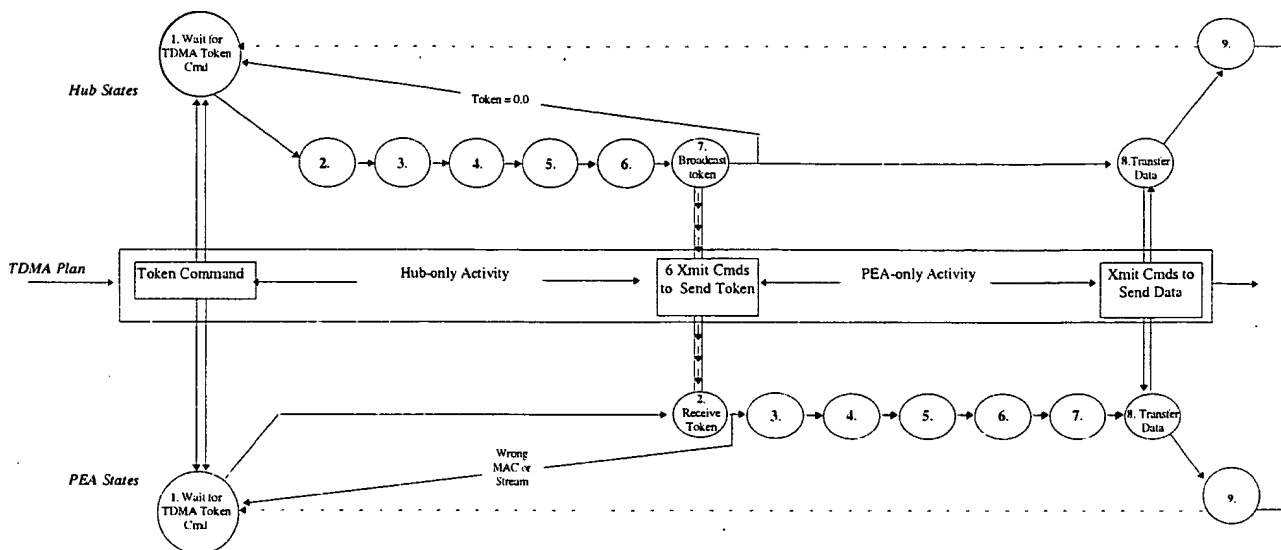


Figure 8: How the PEA and Hub Machines Interact

Figure 8 shows the transmission of a single data block according to the TDMA plan. The figure indicates how a token command (e.g.,  $T_0$  or  $T_1$ ) is processed by the Hub: a token is located; the token is transmitted; and then the data buffer following the token is transmitted. After the transmission of the data, the state machines return to State 1 (Wait for TDMA Token Command), and wait for the next TDMA token command (which occurs in time to the right of the figure).

The individual states for PEA and Hub are shown in the following two figures in relation to their relevant data structures:

- The Hub and PEA both use *descriptor rings* to point into the data buffer to be transmitted. The ring's *header* has fields that include a current pointer, identifying a specific entry in the ring; *flags* (direction of the stream, CRC and interrupt enable) that control the behavior of the stream; a *length* field for the data contained in the attached data blocks, and a receive/transmit *counter* for the stream.
- For a Hub, the *token plan ring* provides entry into a current descriptor ring. There are four token plan rings, one for each type of token ( $T_0$ ,  $T_1$ ,  $T_2$ , or  $T_3$ ). For each token plan ring, there may be many descriptor rings. For any given descriptor ring, however, all the data blocks are the same length.
- For a PEA, the *MAC* and *stream locator* arrays index into one of potentially many descriptor rings. There is one MAC array and one or more stream locator arrays.

### 3.2.1 How Hub Structures Relate to the TDMA Plan

This section describes the Hub's-eye view of the TDMA plan and how the Hub's data structures are used as the plan is executed.

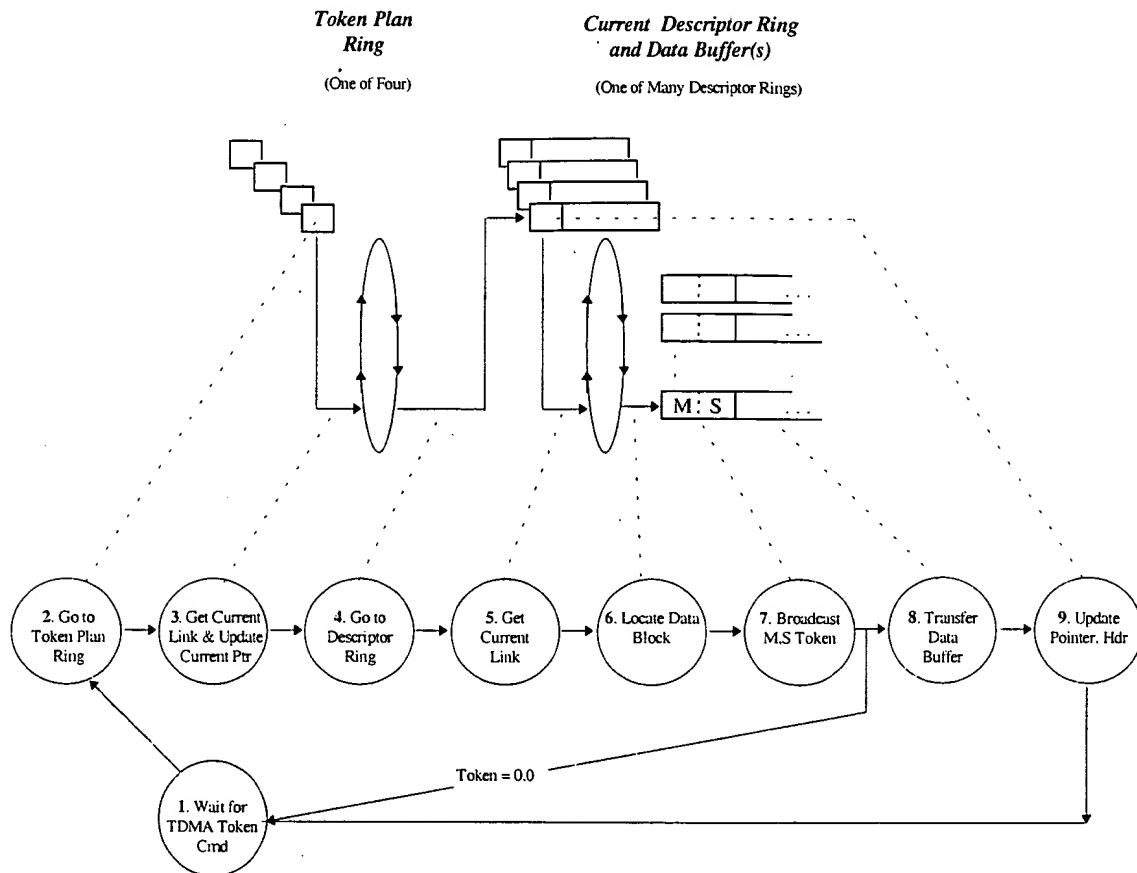


Figure 9: How Hubs Use the Data Structures

#### Notes:

- 1) The Hub hardware has reached a token command in the TDMA plan.
- 2) The token command is one of four types, specifying which token plan ring to use.
- 3) The current link in the token plan ring determines which descriptor ring to use. Follow the pointer (and update the current-link pointer in the token plan ring).
- 4) The descriptor ring selected is one of many.
- 5) Get the current link in the descriptor ring, which points to a data block containing a token and data buffer.
- 6) Get the data block (its first two bytes are the token to send).
- 7) Broadcast the token with its MAC and stream number components.

- 8) Using the data buffer following the token, send or receive data bytes (the direction flag and length are in the ring header).
- 9) Update the current-link pointer from the next-link pointer in the current link. Also update the count value in the descriptor ring header.

### 3.2.2 How PEA Structures Relate to the TDMA Plan

This section describes the PEA's-eye view of the TDMA plan and how the PEA's data structures are used as the plan is executed.

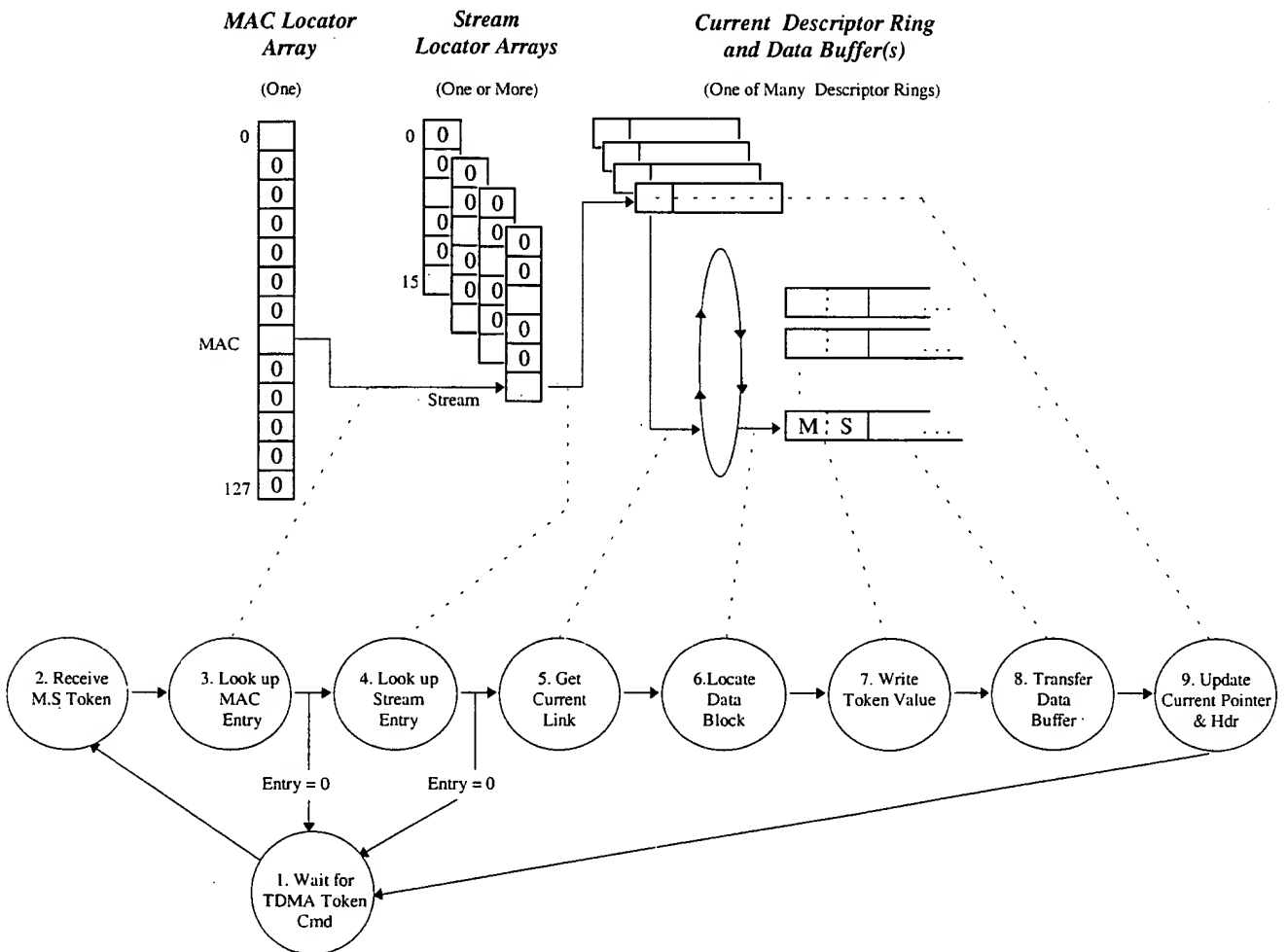


Figure 10: How PEAs Use the Data Structures

Notes:

- 1) The PEA hardware has reached a token command in the TDMA plan. (The PEA pays no attention to which of the four types it is.)
- 2) The PEA receives the MAC and stream numbers broadcast by the Hub.

- 3) Determine which stream locator to use, based on the broadcast MAC number as an index into the MAC array. If the MAC array contains a zero value instead of a pointer at the specified index (i.e., MAC-Locator-Array[MAC]=0), return to State 1.
- 4) Determine which descriptor ring to use, based on the stream number as an index into the stream locator array just found. If the stream locator array contains a zero value instead of a pointer at the specified index (i.e., Stream-Locator[stream]=0), return to State 1.
- 5) Get the current link in the descriptor ring thus specified by the MAC and stream numbers. This link points to a data block containing a token and data buffer.
- 6) Get the data block (its first two bytes will receive the token value).
- 7) Write the MAC and stream numbers into the token part of the data block.
- 8) Using the data buffer following the token, send or receive data bytes (the direction flag and length are in the ring header).
- 9) Update the current-link pointer from the next-link pointer in the current link. Also update the count value in the descriptor ring header

#### 4. Software View of Hardware Setup (Static View)

The common TDMA plan is executed continuously, broadcasting a 6-byte beacon, then a 6-byte  $T_0$  token command, followed by a 4-byte Xmit that broadcasts the token containing the MAC and stream numbers, followed in turn by a pattern of 7 groups, each containing a 6-byte  $T_1$  token command and 20 bytes of Xmit that broadcast 20 bytes from the data block (including the token). Then the plan starts over.

This plan provides for the exchange of two sizes of data blocks: small and large. The small (4-byte) blocks are used exclusively for sending communications about data. The large (20-byte) blocks are used to send data or communications about data.

Figure 11 shows that the  $T_1$  token command and its 20 Xfer bytes can send either large or small data buffers, whereas the  $T_0$  token command and its 4 Xfer bytes can send only small data buffers.

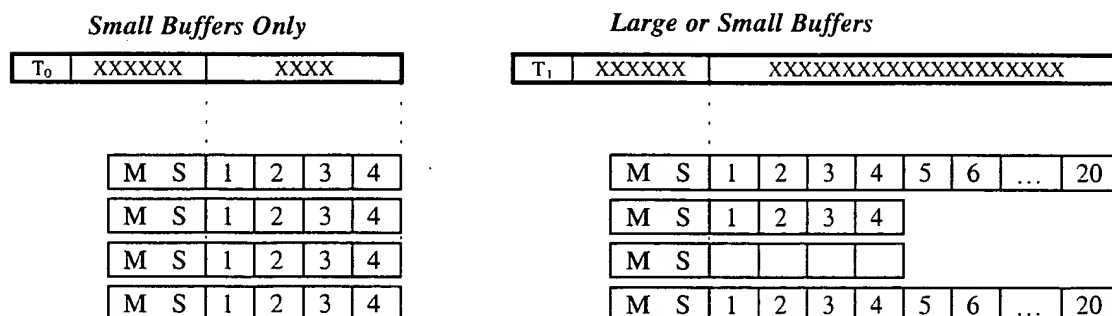


Figure 11: Small and Large Data Buffers

## 4.1 Stream Basics

The Hub and PEAs set up groups of descriptor rings for input and output, depending on the communication direction and length of the data buffer. These rings, associated data blocks, and stream group may be represented in Figure 12. Here, 4-byte Hub-to-PEA data blocks and their stream numbers (8, 9, a) are in mixed order to indicate they are organized by descriptor ring and not in order of stream number. The right half of indicates how this might be displayed in an abstract form with a ring followed by its associated stream numbers, as in Figure 13.

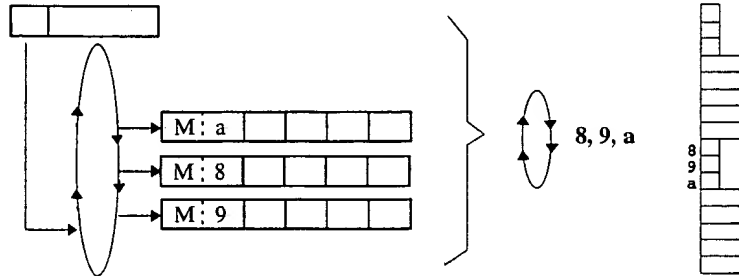


Figure 12: Sample Descriptor Ring and Data Blocks for Hub's Hub-to-PEA Data

## 4.2 How the Hub Sets up Descriptor Rings for Streams

For a Hub, the token ring plans for token types  $T_0$  and  $T_1$  point to structures as shown in Figure 13. Here, the stream numbers are shown next to the descriptor rings. The Schedule columns show the relative frequency with which each associated data block is scheduled by the Hub.

	Schedule	Streams for Small Data Buffers	Schedule	Streams for Large Data Buffers
PEA in to Hub	$T_0$ slower		$T_1$ normal	
	$T_0$ slow			
	$T_1$ normal			
Hub out to PEA	$T_0$ slower		$T_1$ normal	
	$T_1$ normal			

Figure 13: The Hub's Stream-Related Structures

Notes for Figure 13:

- 1) Current-link pointers and associated headers are not shown.
- 2) There is no slow, Hub-out-to-PEA, small ring, as no need was found for it.

### 4.3 How PEAs Set up Descriptor Rings for Streams

The PEA sets up the structures shown in Figure 14. They are pointed to by non-zero elements of the stream locator arrays, which are in turn pointed to from the MAC locator array.

Notes for Figure 14:

- 1) The rings' current-link pointers are shown, but associated headers are not.
- 2) Not all entries in each stream locator array are populated at any one time, but those that are must point to the descriptor ring indicated.
- 1) The PEA's table of stream-related structures shows no difference in scheduling for the various data buffers, because the Hub, not the PEA, schedules frequency.

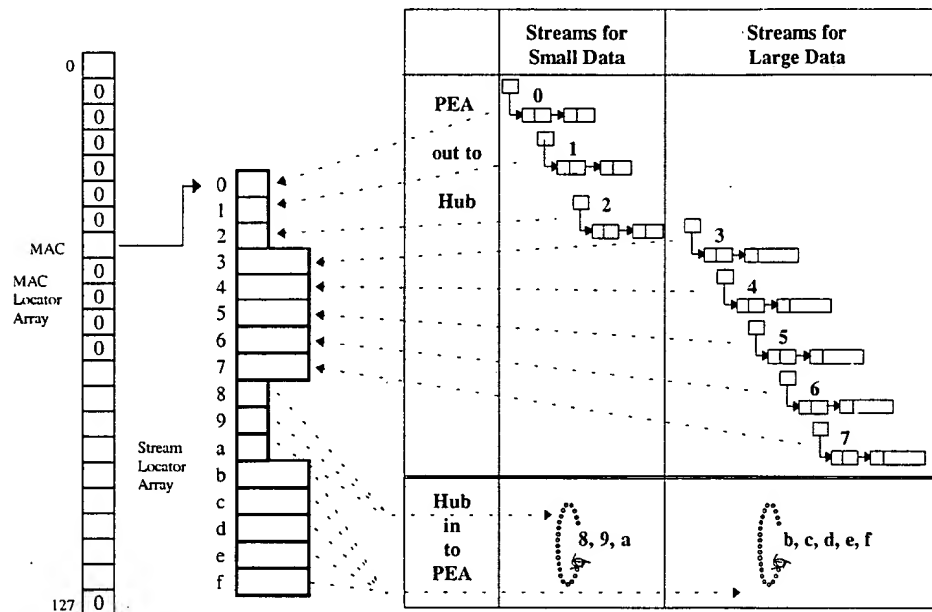


Figure 14: The PEA's Stream-Related Data Descriptor Structures

## 5. Descriptor Rings for Hubs and PEAs (Dynamic View)

## 5.1 The Hub's Dynamic View of the Descriptor Rings

For the Hub, the links of the descriptor rings are taken on and off the chain dynamically, using normal linked-list manipulations. Each link permanently points to a data block in the BodyLAN memory. As links are added (or removed), a ring of descriptors forms and grows (or shrinks).

Hub output rings will consist, at various times, of many entries, one entry, or no entry, as shown in Figure 15.

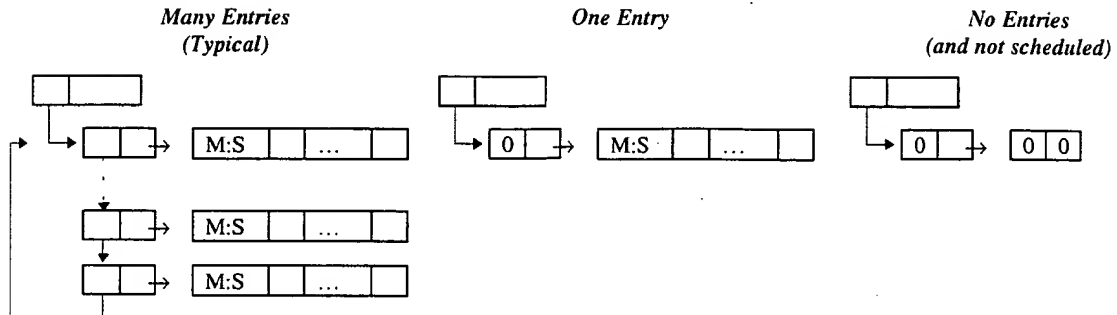


Figure 15: Hub Output Rings

Hub input rings will consist, at various times, of many entries, a single entry, or no entry, as shown in Figure 16.

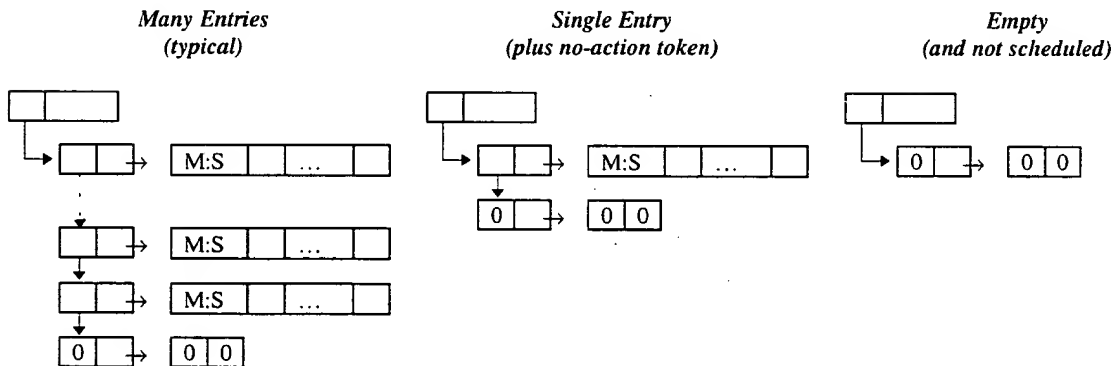


Figure 16: Hub Input Rings

The Hub reuses data blocks (when free and available), reinserting them into the input chain before the last element, maximizing its opportunity to collect input. This design is not optimal, as any scheduling of the no-action token (0, 0) wastes a transfer opportunity, but it has the advantage of minimizing the writes to the hardware memory to add or delete links. Ideally there should be a pair of descriptor rings, one for the tokens, and a second to actually receive the data (set up to be as long as possible, as is done at the PEA).

## 5.2 The PEA's Dynamic View of the Descriptor Rings

For the PEA, each output descriptor ring points only to a single data block in BodyLAN memory, whose contents the hardware sends out whenever the Hub schedules the transmission. When the transport layer calls the link layer driver to send a data buffer, the driver puts the buffer into the extra data block (see Figure 17), and then sets the current-link pointer to point to the extra data block's descriptor link, thus replacing the active descriptor as the only hardware-visible descriptor, effectively updating the ring's single value.

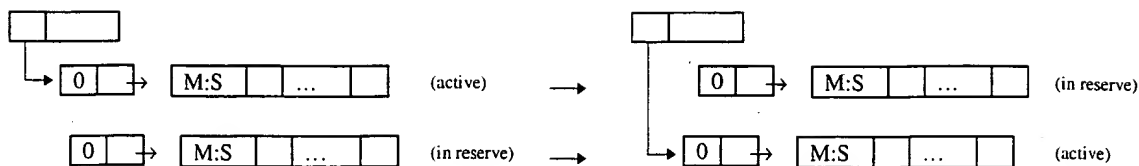


Figure 17: PEA Single-Entry Output Rings

The PEA uses a large, fixed-size receive ring, as shown in Figure 18, below. It fills the ring's data buffers from the first to last, and then (if the software is not timely) fills the last over and over until the software resets the current-link pointer.

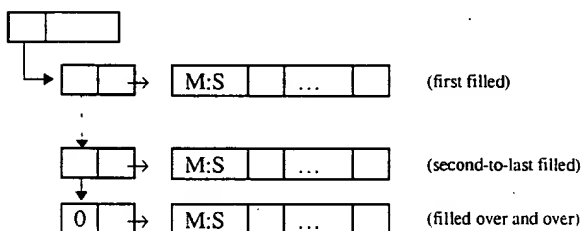


Figure 18: PEA Input Rings

### 5.3 Scheduling

The Hub schedules communication activity according to the TDMA plan. This scheduling may be thought of as a clock with three faces, as shown in Figure 19.

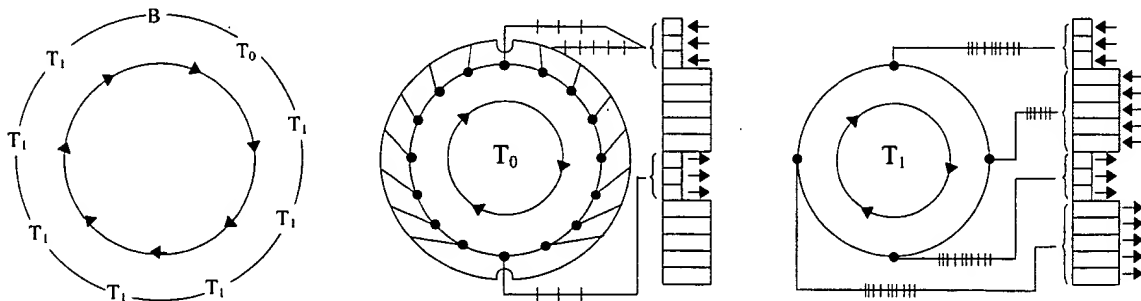


Figure 19: Scheduling of Communication Activity

The TDMA cycles through the Beacon,  $T_0$ , and seven  $T_1$  commands as shown at the left of Figure 19. When The  $T_0$  command is reached, one of the rings associated with  $T_0$  is scheduled, as shown in the middle of the figure: here, the top and bottom activities are both scheduled to be slow in and slow out, respectively, and the other 14 activities are scheduled more slowly. As each  $T_1$  command is reached on the left-most dial, the cycle shown at the right of the figure is advanced one item, with each activity being scheduled at a normal rate. Each advance schedules whatever is next in the sequence—an incoming or outgoing ring (large or small).

The link layer transport uses small blocks to coordinate data transfer and attachment. The Hub polls PEAs for attachment requests, for data ready, and for ACKs (in increasing frequency). The link layer driver, though it does not officially “understand” such uses of small communication blocks, must and does understand the need to provide multiple descriptor rings—normal, slow, and slower—for small data blocks so that they can be scheduled at different frequencies.

At the Hub, the two token plan rings for  $T_0$  and  $T_1$  commands point to groups of rings organized by scheduling frequency and size of data block. The  $T_0$  ring points to small input and output rings scheduled for slower transmission, and to small input rings scheduled for slow transmission (but not as slow). Their relative frequency is controlled by inserting multiple links for each ring, with different frequencies. Ideally, the slower rings might be scheduled even more sparsely than they are, but there is no way to fine-tune frequencies given the current memory-map design.

The  $T_1$  token plan ring points to four normal-speed rings (small and large in, small and large out). Normally, when these rings are all filled (see Figure 20), they are all scheduled in sequence.

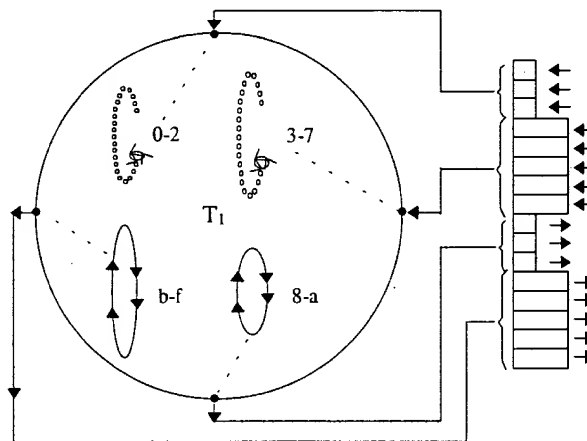


Figure 20: Scheduling When All Rings Are Filled

For efficiency, these links are manipulated so they always point to the subset of non-empty descriptor rings. A scheduling routine is called when adding a first data block into an empty ring or removing a final data block from a (non-empty) ring. These links always point to non-empty rings, and in the special case of four empty rings, they all point to slow or slower input rings instead of letting the time go to waste. Figure 21 shows what happens when various combinations of these rings are empty—but only for a few of the total 16 possibilities.

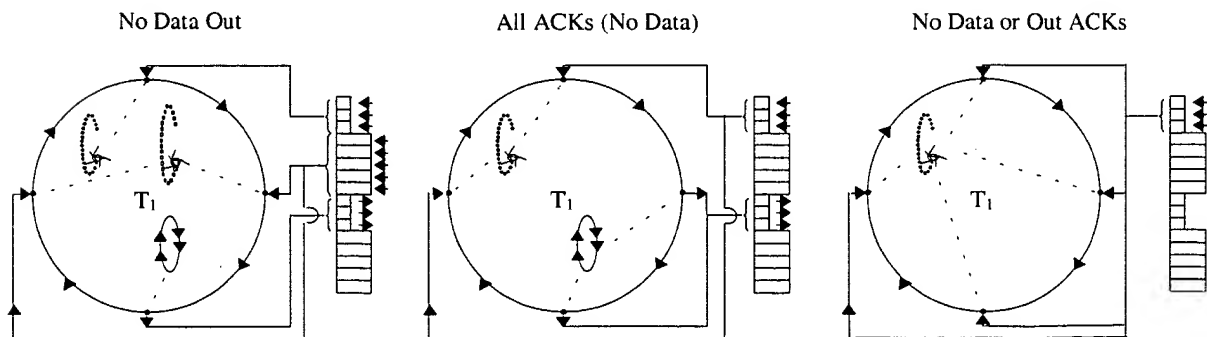


Figure 21: Scheduling for Various Empty Rings

## Exhibit B



# BodyLAN Hardware/Software Memory Interface

---

Document number:      BodyLAN-13-06  
Last modified:        December 12, 1997  
Author:                Bob Donaghey and Rod Owen

## Table of Contents

<i>Introduction</i> _____	2
<i>Layout of Hardware Memory</i> _____	2
<i>Execution of Token(N) and Xmit(KK) Commands</i> _____	4
At the Hub _____	4
Simultaneously at a PEA _____	7

## Introduction

This document provides notes describing the Altera memory layout and how it is used by the Hub and PEA to execute the TDMA commands **Token(N)** and **Xmit(K)**.

It assumes that the reader is already familiar with terminology such as *TDMA*, *token plan ring*, *descriptor ring*, *stream*, *token*, etc., described elsewhere in the BodyLAN documentation and summarized in *BodyLAN Link Layer Driver Architectural Details*, Document BodyLAN-32-01.

For more explanation of the symbols for the software structures and the interaction of the PEA and Hub finite state machines with them, also see BodyLAN-32-01.

Note that the finite state machine(s) used to describe the interaction of the memory structures for the PEA and Hub have been generalized somewhat, in that the software implements similar machines but with slightly different states.

## Layout of Hardware Memory

The Hub and PEA behaviors are different, but either hardware can assume the other's role, It is assumed that:

- They share a common TDMA plan.
- There is a software-set flag in the globals at the top of memory that indicates to hardware whether to act as the Hub or as a PEA.

The hardware memory for the Hub and PEA is shown in Figure 1. For architectural purposes, this memory may be considered to consist of multiple rows, each 256 bytes wide. The schematic representations of pointers, headers, arrays, linked lists, and data blocks shown at the left of the figure are explained more fully in text following it.

The character symbols displayed in the memory map and used in the discussion in this document are briefly explained in the following table. The use of "<<8" or "<<4" indicates that the value is left-shifted eight or four bytes to obtain an address.

Symbol	Brief Explanation
<b>AA<sub>N</sub></b>	Token locator 0...3: 1 <b>AA<sub>N</sub></b> is the address of <b>DD</b> and <b>Next AA<sub>N</sub></b> in one of four token plan rings.
<b>FB</b>	<b>B&lt;&lt;16</b> is the highest nibble of the descriptor ring block pointers, <b>F</b> is flags (reserved).
<b>DD</b>	<b>DD&lt;&lt;8</b> = the address for descriptor ring header.
<b>HH</b>	The second byte of descriptor ring data block address.
<b>II</b>	The offset, OR'd with <b>DD&lt;&lt;8</b> to provide address of descriptor in descriptor ring
<b>LL</b>	The low byte of descriptor ring data block address.
<b>MM</b>	Mac Number (00..7F), 1 <b>MM</b> is address of <b>RR</b> (or 0).
<b>N</b>	Token register locator in TDMA plan, <b>N</b> is index of <b>AA<sub>N</sub></b> .
<b>RR</b>	PEA stream Index. <b>RR &lt;&lt; 4 + 0S</b> is address of <b>DD</b> (or 0).
<b>0S</b>	Stream number (0..15). <b>MM</b> and <b>0S</b> make up a token.

Table of Named Symbols Used in this Document

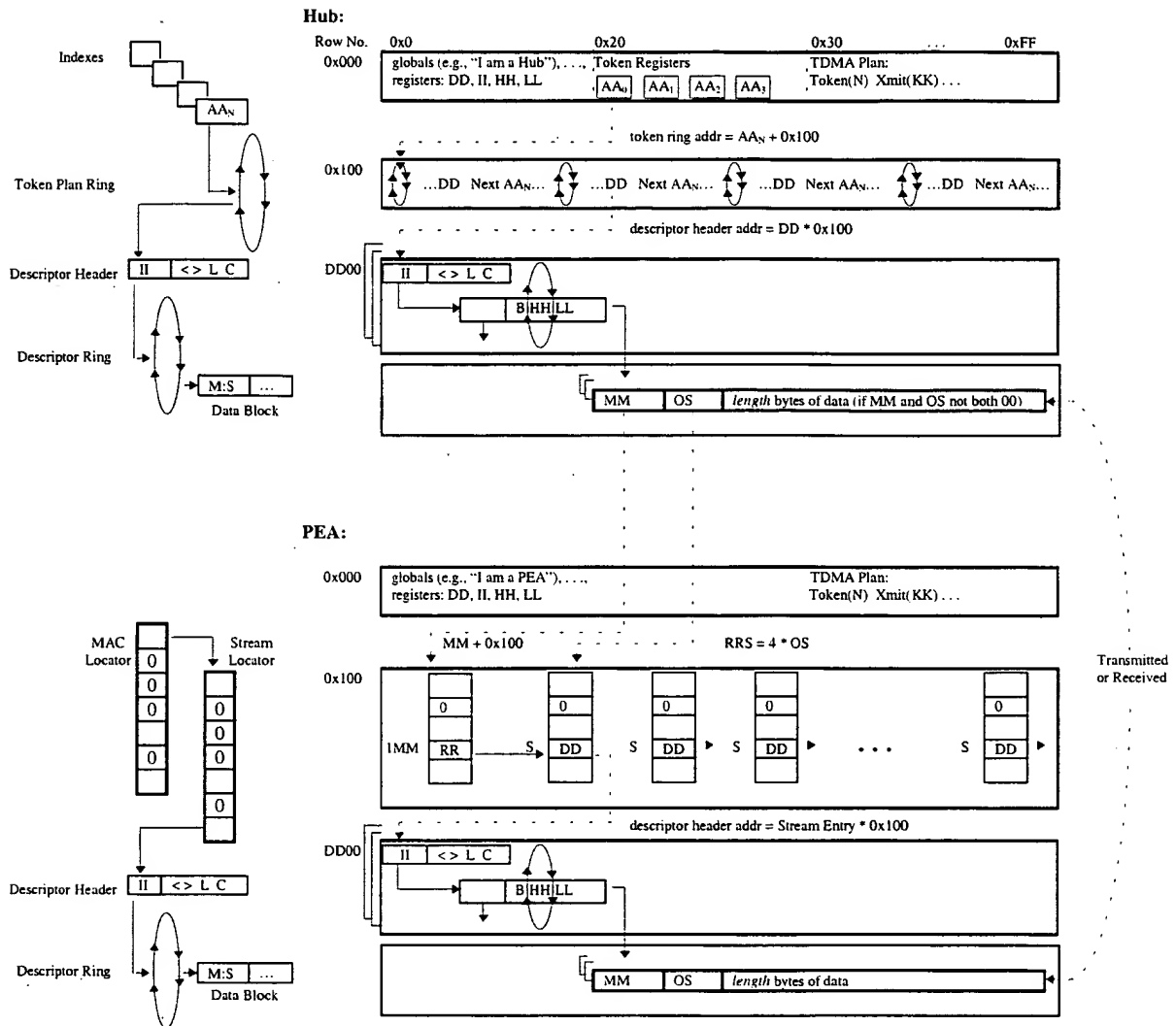


Figure 1: Hardware Memory Layout for Hub and PEA

Figure 1 relates, for a Hub (in the top half of the figure) and a PEA (in the bottom half), the software structures (shown on the left of the figure) and the hardware memory layout.

The software structures and their interaction are described more fully in *BodyLAN Link Layer Driver Architectural Details*, Document BodyLAN-32-01, but they also appear, below, in Figure 2 and Figure 4, in relation to the state diagrams for the Hub's and PEA's memory access and data transfer.

The hardware instantiates these structures in 256-byte blocks of memory, as shown in the figure. Notice that the Hub uses address space beginning at 0x20 in Row 0x000 to contain the token registers, and Row 0x100 to contain token planning rings, whereas the PEA does not use the space at 0x20 but uses Row 0x100 to contain the MAC and Stream locator arrays. Note also that there are multiple "DD00" rows; and that the (multiple) data blocks (of token plus data buffer) may be anywhere within a row of memory, but may not span rows.

## Execution of Token(N) and Xmit(KK) Commands

The actions related to the memory structure and numbered as the states appear in the finite state machine diagram are presented in the following format:

**State # Action:** (a) **Text from state diagram.** (b) Summary of the action. More explanation.

- Notes concerning the action.

### At the Hub

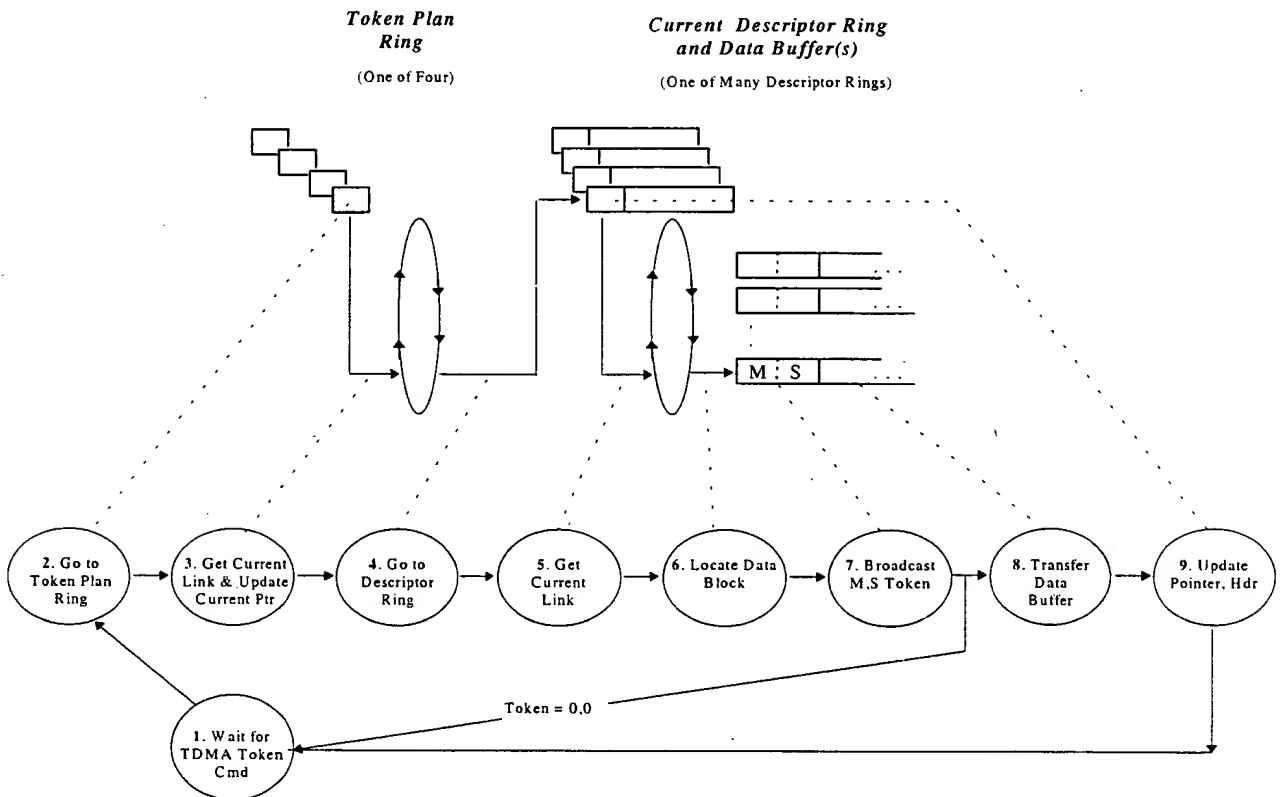


Figure 2: State Diagram for Hub's Memory Access and Data Transfer

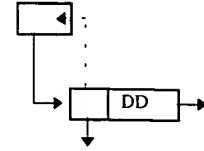
The following states are cross-referenced as Hub-1, Hub-2, etc., from other locations in this document.

- 1) Action: **Wait for token command** in TDMA plan.
- 2) Action: **Go to token plan index register.** Find **Token(N)** in the TDMA plan that begins at address 0x30. From the register specified by N, fetch register value AA.
  - The Hub TDMA plan contains ... **Token(N)** , **Xmit(KK)**, ... and the hardware has reached the **Token(N)** command. The TDMA plan is generated by software, but is never changed dynamically, so N and KK are fixed. Also, the **Token(N)** and **Xmit(KK)** are not meant to specify how the TDMA plan is written, but only to provide a shorthand summary of its contents.

- There are four registers for the TDMA plan to use, so N must be in the range 0..3.

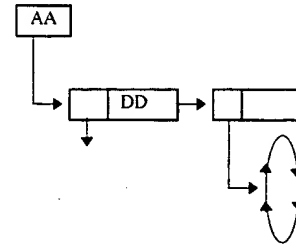
1) Action: **Get current link to token planning ring and update register.** Update the register specified by N to hold the value found at address  $0x101 + AA$ .

- In the register specified by N is the value AA used in step Hub-2. At location  $1AA++$  (i.e.,  $AA + 0x101$ ) is the next value for the register specified by N.



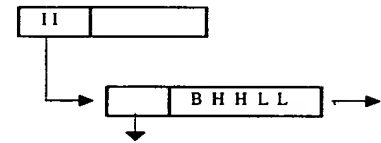
2) Action: **Go to descriptor ring.** At location  $1AA$  (i.e.,  $AA + x100$ ), fetch value DD.

- The DD values will be manipulated by software dynamically to send the hardware to the correct row, because the TDMA plan must know how large the following data block can be, but must not specify the direction of data flow or the use (i.e., MAC and stream numbers) of the block. When the TDMA plan is crafted, the numbers N will be chosen (in the range 0 to 3) to permit software to dynamically assign the DD values from the correct group of descriptor rings for each KK-length data block without having to know the pattern of interleaving of the large and small data blocks.



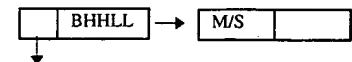
1) Action: **Get the current link.** Read II in the descriptor header at the beginning of the row, then form the address DDII (i.e.,  $DD \ll 8 \vee II$ ).

- The value DD in location  $1AA$  identifies a row of memory (i.e.,  $DD \ll 8 + 00..FF$ ). The beginning of the row address is given as DD00 Figure 1. This row is guaranteed to contain a descriptor ring, which is a structure known to both the software and hardware to consist of a descriptor header followed by a number of descriptors. The software guarantees to hardware that any row accessible from  $1AA$  for any **Token(N)** in the TDMA plan is a descriptor ring.

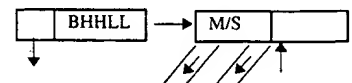


1) Action: **Locate data block.** At the descriptor address, read LL, HH and FB. From values B (the low nibble of FB), HH, and LL, form the address of the memory location BHHLL (i.e.,  $B \ll 16 \vee HH \ll 8 \vee LL$ ) which forms a 20-bit memory address.

- The three bytes of the address are stored in little-endian format, with low byte first.
- It should be expected that software will use only as much memory as it absolutely needs, so it is highly likely that B will be 0. It is the responsibility of software to manage the memory in an error-free manner.



1) Action: **Broadcast M/S token.** Send the MAC and stream numbers MM and OS found at the memory location BHHLL out to the PEAs using "reliable transmission" (whose exact definition is strictly up to hardware). If both MAC and stream number values are 00, then sleep through the following **Xmit(K)** instruction in the TDMA plan, because no PEA will respond to this special token value.



- In the memory at location BHHLL, Hub software has placed a 2-byte token (MM, OS) followed by the corresponding data buffer (which is full for sending and empty for receiving). The least 7 bits of MM and the least 4 bits of OS are significant.

- These two values comprise the *token* and are collectively 11 bits in length. The hardware guarantees that 7-bit MAC and 4-bit stream values are delivered at the PEA, so the PEA needs to allocate 128 bytes for the MAC locator array, and 16 bytes for each stream locator array addressed through it.

At this point, the Hub waits until the PEA accomplishes its actions for States 3 through 7 and is ready to exchange the data. (See Figure 3, and PEA-3 through PEA-8 in the following section.)

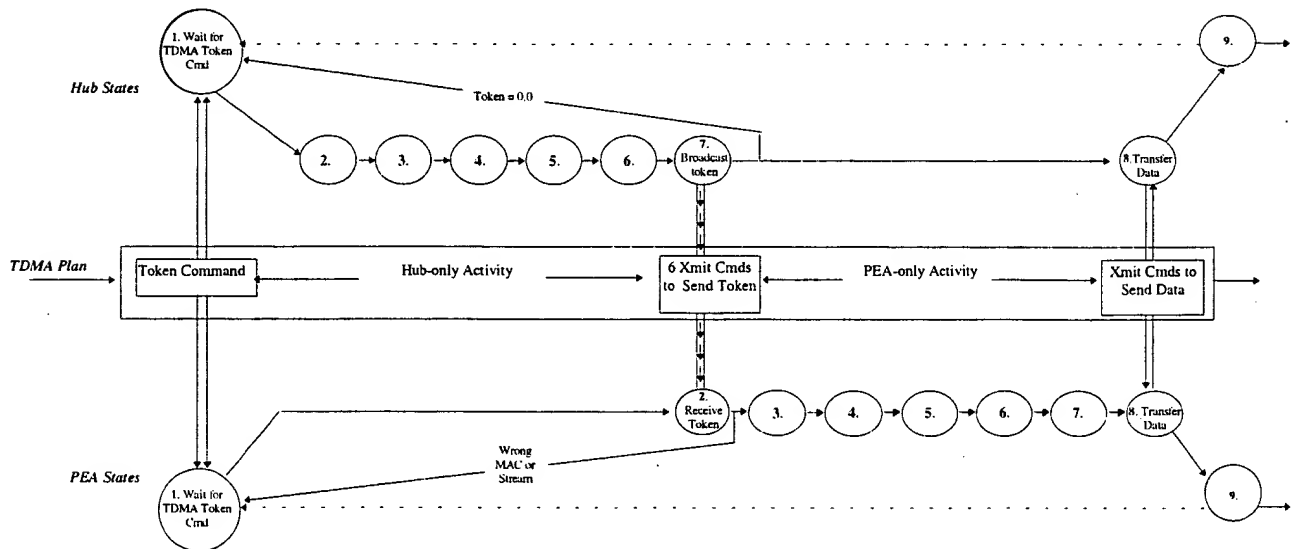
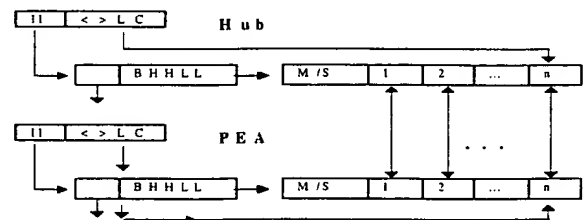


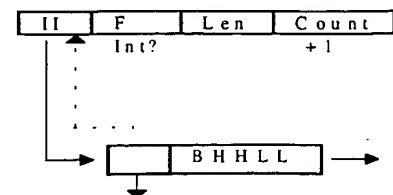
Figure 3: How the PEA and Hub Machines Interact

- Action: **Transfer the data buffer.** Save this location and read the Flags in the descriptor header at the beginning of the row DD00 to determine the direction of the Xmit. Then wait for the **Xmit(KK)** activity as the PEA needs time to prepare (as just mentioned above).

- The next byte beyond the MAC and stream numbers is the correct starting place for the TDMA Xmit activity to follow.
- Software guarantees that the length of the current data buffer is compatible with the KK in the TDMA plan, which is to say no greater than KK. The minimum legal value for length is 1 and the maximum legal value is the lesser of KK and 0xFE.



- Action: **Update the pointer and header in the descriptor ring.** If the Next-II value is not 00, then copy the Next-II to the II position at the beginning of the stream-header at DD00, thus moving the hardware to the next element of the linked list. Then (in either case) increment the event counter in the descriptor header so that software will see that the data transfer has taken place (software keeps its own counter and compares the two counters to see how many events it is behind



the hardware). Then set the global, software-visible registers DD, II, HH, and LL to 00 to let software know that the task is complete and the software may modify pointers and free memory, etc., without getting in the hardware's way. Generate an interrupt (if possible) to software as the final step, if indicated in the flag byte in the stream-header at the beginning of row DD00. (The direction indicator for the row is also in this byte. The other six bits of this flag byte are reserved.)

- The current ring descriptor at location DDII contains a Next-II value (the descriptors form a linked list), and the ring header contains an event counter. The Next-II value, as a special case, may be 00, which, although not a valid II value, instructs the hardware not to update the stream-header, but to stay at the same descriptor until software changes the II in the stream-header.

Under some conditions, the hardware may be instructed to sleep through the Xmit activity. If this happens, the hardware should skip any remaining Xmit steps and resume activity at the next instruction following the **Xmit(KK)** in the TDMA plan.

### Simultaneously at a PEA

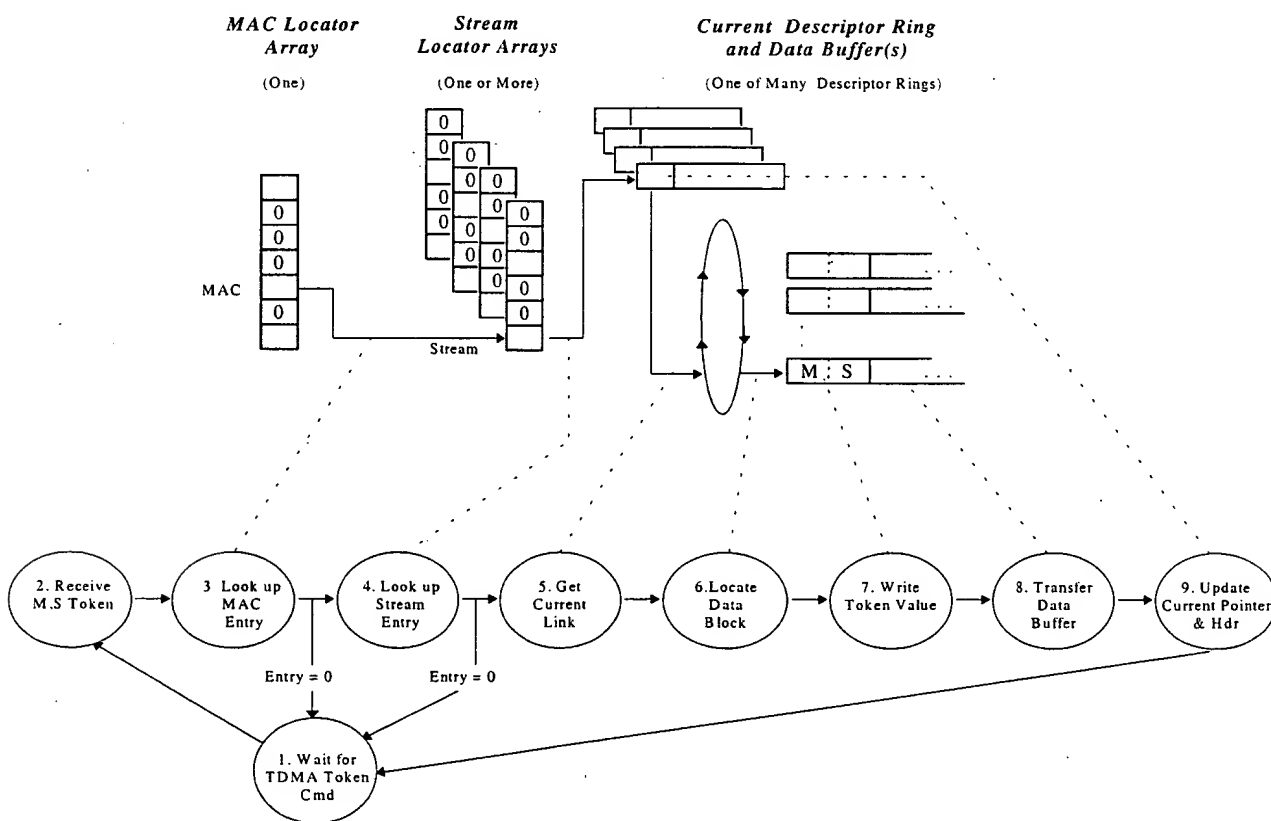
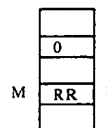


Figure 4: State Diagram for PEA's Memory Access and Data Transfer

Some of the following states are cross-referenced as PEA-3, PEA-4, etc., for example, from the note before Hub-8, above.

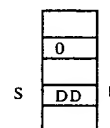
- 1) Action: **Wait for token command.** In the TDMA plan.
- 2) Action: **Receive M/S token.** Wait for the Hub to transmit the MAC and stream number values MM and OS using "reliable transmission" (whose exact definition is strictly up to hardware). Decode the MM and OS numbers if possible, or sleep through the **Xmit(K)** instruction in the TDMA plan if not.
  - The most significant bit of MM is guaranteed by hardware to be 0, as are the four most significant bits of OS.
  - The value N in the **Token(N)** command is not used by the PEA.

- 1) Action: **Look up MAC entry.** Determine which stream locator to use, based on the broadcast MAC number as an index into the MAC locator array, which starts at 0x100. Fetch from address 1MM (i.e.,  $MM \vee 0x100$ ) the value RR. If this value is 00, then sleep through the **Xmit(K)** instruction in the TDMA plan.



- Since MM is a 7-bit number, 1MM takes on values between 100 and 17F.

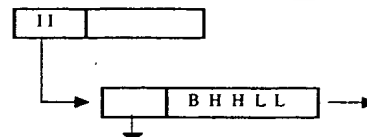
- 1) Action: **Look up stream entry.** Fetch from address RRS (i.e.,  $RR \ll 4 \vee OS$ ) the value DD. If this value is 00, then sleep through the **Xmit(K)** instruction in the TDMA plan.



- The  $1/16^{\text{th}}$ -row RR contains a set of stream-indexed DD values.
- Since OS is a 4-bit number, RRS takes on values between  $RR \ll 4$  and  $RR \ll 4 + 0F$ .
- The Hub will sleep through the **Xmit(K)** if both MM and OS are 00, so the software guarantees that either the value at 1MM is 00 or the value at 1MM is an RR for which the value at address RR0 is 00 (i.e., the PEA will also sleep through the **Xmit(K)** when the HUB does).
- There are no further conditional checks by the PEA..

- 2) Action: **Get the current link.** As the Hub did in step Hub-5, read II in the descriptor header at the beginning of the row, then form the address DDII (i.e.,  $DD \ll 8 \vee II$ ).

- The value DD in location RRS identifies a row of memory (i.e.,  $DD \ll 8 + \text{offset } 00..FF$ ). The beginning-of-row address is given as DD00 in this discussion. This row is guaranteed to contain a descriptor ring, a structure known to both the software and hardware to consist of a descriptor header followed by a number of descriptors. The software guarantees to the hardware that any row accessible from  $(1/16^{\text{th}})$ -row RR, with RR obtained from step Pea-2, is a descriptor ring. This is identical in detail to the descriptor row found by the Hub in step Hub-4.

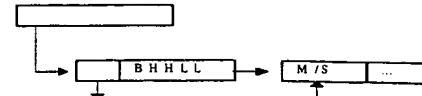


- 1) Action: **Locate data block.** At the address DDII at this descriptor address, read LL, HH and FB. From values B (the low nibble of FB), HH, and LL, form the address of the memory location BHHLL (i.e.,  $B \ll 16 \vee HH \ll 8 \vee LL$ ) which forms a 20-bit memory address.

- The three bytes of the address are stored in little-endian format, with low byte first.

- 2) Action: **Write token value.** Write the MAC and stream numbers MM and OS into the memory at location BHHLL.

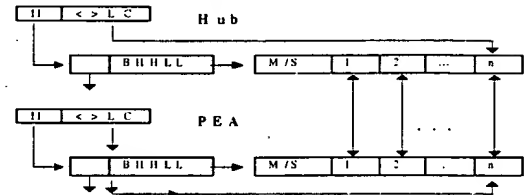
- In the memory at location BHHLL is a data structure consisting of the location of a two-byte token (MM, OS), followed by the corresponding data buffer that has been filled in by software for sending, or is assumed by hardware to be empty for receiving).



The PEA has now "caught up" with the Hub, which is still waiting to accomplish the actions at State 8. Now, they both begin transferring the data simultaneously.

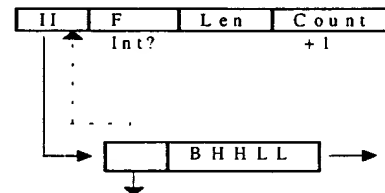
- 1) Action: **Transfer data buffer.** Save this location and read the flags in the descriptor header at the beginning of the row to determine the direction of the Xmit. Then wait for the Xmit(KK) to follow.

- The next byte beyond the MAC and stream numbers is the correct place for the TDMA Xmit activity to follow. In the stream-header at the beginning of row DD00, the flag byte indicates the direction of transfer of the data buffer.



- Using the memory location and transfer direction, read *length* out of the descriptor ring header and Xmit *length* bytes from the send buffer to the receive buffer. That is, if sending, send *length* bytes found at the memory location, and if receiving, receive *length* bytes into the memory location
- Software guarantees that on both ends *length* is no greater than KK for correctly transmitted tokens. A value greater than KK may be used by hardware to reject the token. A value of 0 is illegal: software guarantees that *length* is  $\geq 1$ .
- The TDMA plan contains ... **Token(N)**, **Xmit(KK)**, ... and the hardware has reached the **Xmit(KK)** command.

- 1) Action: **Update the pointer and header in the descriptor ring.** If the Next-II value is not 00, then copy the Next-II to the II position at the beginning of the stream-header at DD00, thus moving the hardware to the next element of the linked list. Then (in either case) increment the event counter in the descriptor header so that software will see that the data transfer has taken place (software keeps its own counter and compares the two counters to see how many events it is behind the hardware). Then set the global, software-visible registers DD, II, HH, and LL to 00 to let software know that the task is complete and that it is safe for the software to modify pointers and free memory, etc., without getting in the hardware's way. Generate an interrupt (if possible) to software as the final step if indicated in the flag byte in the stream-header at the beginning of row DD00. (The direction indicator for the row is also in this byte. The other six bits of this flag byte are reserved.)



- The current ring descriptor at location DDII contains a Next-II value (the descriptors form a linked list), and the ring header contains an event counter. The Next-II value, as a special case, may be 00, which, although not a valid II value, tells the hardware not to update the stream-header, but to stay at the same descriptor until software changes the II in the stream-header.

# Exhibit C.txt

```

/* --- Simulate_Hub() --- */
Simulate_Hub()
{
    TDMA_INSTRUCTION opcode;
    int length = 0, position;
    TOKEN_PLAN_RING *token_plan_ring;
    DESCRIPTOR_ROW *row;
    PACKET_DESCRIPTOR *p;
    BYTE *packet, *packet_start, arg;
    BOOLEAN sending_token, sending_beacon, transmitting, no_actual_packet;

    sending_token = sending_beacon = no_actual_packet = FALSE;

    Init_TDMA(0);

    fprintf(stderr, "Hub %02x%02x%02x%02x starting.\n",
        Control->id4, Control->id3, Control->id2, Control->id1);

    REPEAT
        switch(opcode = Next_Instruction(&arg))
        { case BEACON:
            { BYTE *data = Beacon_String + NUM_BEACON_BYTES;

                Sync_To_TDMA();
                position = TDMA_PC - Control->tdma_plan;
                data[HUB_ID_1] = Control->id1;
                data[HUB_ID_2] = Control->id2;
                data[HUB_ID_3] = Control->id3;
                data[HUB_ID_4] = Control->id4;
                data[TDMA_INDEX_HIGH] = (position >> 8) & 0xff;
                data[TDMA_INDEX_LOW] = position & 0xff;
                packet = Beacon_String;
                length = NUM_BEACON_BYTES + BEACON_DATA_SIZE;
                sending_beacon = transmitting = TRUE;
                break;
            }

            case TOKEN:
                Sync_To_TDMA();
                token_plan_ring = (TOKEN_PLAN_RING *)
                    (Memory + (Control->token_register[arg] | 0x100));
                Control->token_register[arg] = token_plan_ring->next_AA;
                if (! (Control->DD = token_plan_ring->DD))
                    { length = 0;
                        break;
                    }
                row = &Row[token_plan_ring->DD];
                p = (PACKET_DESCRIPTOR *)(((BYTE *) row) +
                    (Control->II = row->II));
                packet = Memory +
                    (((p->FB & 0xf) << 16) |
                    ((Control->HH = p->HH) << 8) | (Control->LL = p->LL));
                length = NUM_TOKEN_BYTES;
                sending_token = TRUE;
                if (! packet[0] && ! packet[1])
                    no_actual_packet = TRUE;
                transmitting = row->flags & DIRECTION;

                #if 1
                packet_start = packet+2;
                if (transmitting)
                    fprintf(stderr, "token is %x,%x for packet \"%s\", length = %d\n",
                        Control->id4, Control->id3, packet_start, length);
                #endif
            }
        }
    }
}

```

```

                                Exhibit C.txt
        packet[0], packet[1], packet_start, row->length);
    else
        fprintf(stderr, "token is %x,%x for receiving packet, length = %d\n",
            packet[0], packet[1], row->length);
#endif
        break;

    case XMIT:
        ++arg;
        while(arg--)
            Doze(TDMA_DOZE);
        if (length)
            { if (sending_token || transmitting)
                Broadcast(*packet++);
              else
                Receive(packet++);

            if (! --length)
                { if (sending_beacon)
                    sending_beacon = FALSE;
                  else
                    if (sending_token)
                        { length = no_actual_packet ? 0 : row->length;
                          sending_token = no_actual_packet = FALSE;
                        }
                    else
                        { if (p->next_II)
                            row->II = p->next_II;
                          row->counter++;
                          Control->DD = Control->II = Control->HH = Control->LL =
0;
                        }
                    #if 1
                    if (! transmitting)
                        fprintf(stderr, "Got packet:  \"%s\\\"\\n", packet_start);
                    #endif
                                if ((row->flags & INTERRUPT) && Interrupt_Routine)
                                    (*Interrupt_Routine)();
                                }
                            }
                        }
                    break;

                default:
                    printf("WHOA NELLIE!!!  Bad instruction in tdma plan!\\n");
                    exit(1);
                }
            ETERNALLY;
        }
    }

```

```

/* --- Simulate_Pea() --- */
Simulate_Pea()
{
    TDMA_INSTRUCTION opcode;
    int length;
    DESCRIPTOR_ROW *row;
    PACKET_DESCRIPTOR *p;
    BYTE *packet, *packet_start, mac, stream, RR, arg,
        beacon_buffer[NUM_BEACON_BYTES], token_buffer[NUM_TOKEN_BYTES];
    BOOLEAN reading_token, reading_beacon, transmitting;

    fprintf(stderr, "Pea starting, looking for hub...\\n");
    sync_with_hub:

```

```

                                Exhibit C.txt
Init_TDMA(Wait_For_Beacon() + BEACON_DATA_SIZE);
Control->status = FULL_SYNC;
fprintf(stderr, "Synced with hub %02x%02x%02x%02x.\n",
           Control->hub_id4, Control->hub_id3, Control->hub_id2, Control->hub_id1);

reading_token = reading_beacon = FALSE;
length = 0;
REPEAT
    switch(opcode = Next_Instruction(&arg))
    { case BEACON:
        Sync_To_TDMA();
        packet = beacon_buffer;
        length = NUM_BEACON_BYTES + BEACON_DATA_SIZE;
        reading_beacon = TRUE;
        transmitting = FALSE;
        break;

        case TOKEN:
            Sync_To_TDMA();
            packet = token_buffer;
            length = NUM_TOKEN_BYTES;
            reading_token = TRUE;
            transmitting = FALSE;
            break;

        case XMIT:
            ++arg;
            while(arg--)
                Doze(TDMA_DOZE);
            if (length)
            { if (transmitting)
                Broadcast(*packet++);
              else
              { if (! Receive(packet++))
                  { fprintf(stderr, "Hub is MIA! De-syncing!\n");
                    Control->status = 0;
                    goto sync_with_hub;
                  }
                }

            if (! --length)
            { if (reading_beacon)
                { if (! strnequ(Beacon_String, beacon_buffer,
                               NUM_BEACON_BYTES))
                    { fprintf(stderr, "Bad Beacon! De-syncing!\n");

#if 0
                    { int index;

                        fprintf(stderr, "(I saw: ");
                        for (index = 0; index < NUM_BEACON_BYTES; index++)
                            fprintf(stderr, " %02x", beacon_buffer[index]);
                        fprintf(stderr, ")\n");
                    }
#endif
                    Control->status = 0;
                    goto sync_with_hub;
                }

                reading_beacon = FALSE;
            }
            else

```

```

                                Exhibit C.txt
        if (reading_token)
        { mac = token_buffer[0];
          stream = token_buffer[1];

#if 1
fprintf(stderr, "Got token:  %x,%x\n", mac, stream);
#endif

          if (! (RR = Memory[0x100 | mac]) ||
              ! (Control->DD = Memory[(RR << 4) | stream]))
          { length = 0;
            break;
          }
          row = &Row[Control->DD];

          p = (PACKET_DESCRIPTOR *)(((BYTE *) row) +
                                     (Control->II = row->II));
          packet = Memory +
                    (((p->FB & 0xf) << 16) |
                     ((Control->HH = p->HH) << 8) |
                     (Control->LL = p->LL));
          *packet++ = mac;
          *packet++ = stream;
          packet_start = packet;
          length = row->length;
          transmitting = row->flags & DIRECTION;
          reading_token = FALSE;

#if 1
if (transmitting)
    fprintf(stderr, "Sending packet \"%s\"\n", packet_start);
#endif

        }
        else
        { if (p->next_II)
          row->II = p->next_II;
          row->counter++;
          Control->DD = Control->II = Control->HH = Control->LL =
0;
#if 1
if (! transmitting)
    fprintf(stderr, "Got packet:  \"%s\"\n", packet_start);
#endif

          if ((row->flags & INTERRUPT) && Interrupt_Routine)
              (*Interrupt_Routine)();
        }
    }
    break;

    default:
        printf("WHOA NELLIE!!!  Bad instruction in tdma plan!\n");
        exit(1);
    }
    ETERNALLY;
}

```

```

/* --- Stream Assignment --- */
/* STREAM NUMBERS == CH numbers plus base */
#define CHANNEL_MASK      7
#define PEA_SS_BASE      0
#define HUB_SS_BASE      8
#define HUB_OUT_STREAM(ch)      ((ch) + HUB_SS_BASE)
#define HUB_IN_STREAM(ch)       ((ch) + PEA_SS_BASE)
#define PEA_OUT_STREAM(ch)      ((ch) + PEA_SS_BASE)

```

```
#define PEA_IN_STREAM(ch)
#define STREAM_CHANNEL(str)
```

```
Exhibit C.txt
                ((ch) + HUB_SS_BASE)
                ((str) & CHANNEL_MASK)
```